Live and Trustworthy Forensic Analysis of Commodity Production Systems

L. Martignoni[1], A. Fattori[2], R. Paleari[2], and L. Cavallaro[3]

[1] Università degli Studi di Udine, Italy
lorenzo.martignoni@uniud.it
[2] Università degli Studi di Milano, Italy
{aristide,roberto}@security.dico.unimi.it
[3] Vrije Universiteit Amsterdam, The Netherlands
sullivan@few.vu.nl

Abstract. We present HyperSleuth, a framework that leverages the virtualization extensions provided by commodity hardware to securely perform live forensic analysis of potentially compromised production systems. HyperSleuth provides a trusted execution environment that guarantees four fundamental properties. First, an attacker controlling the system cannot interfere with the analysis and cannot tamper the results. Second, the framework can be installed as the system runs, without a reboot and without loosing any volatile data. Third, the analysis performed is completely transparent to the OS and to an attacker. Finally, the analysis can be periodically and safely interrupted to resume normal execution of the system. On top of HyperSleuth we implemented three forensic analysis applications: a lazy physical memory dumper, a lie detector, and a system call tracer. The experimental evaluation we conducted demonstrated that even time consuming analysis, such as the dump of the content of the physical memory, can be securely performed without interrupting the services offered by the system.

1 Introduction

Kernel-level malware, which compromise the kernel of an operating system (OS), are one of the most important concerns systems security experts have to fight with, nowadays [1]. Being executed at the same privilege level of the OS, such a malware can easily fool traditional analysis and detection techniques. For instance, Shadow Walker exploits kernel-level privileges to defeat memory content scanners by providing a de-synchronized view of the memory used by the malware and the one perceived by the detector [2].

To address the problem of kernel-level malware and of attackers that are able to obtain kernel-level privileges, researchers proposed to run *out-of-the-box* analyses by exploiting virtual machine monitor (VMM), or hypervisor, technology. In such a context, the analysis is executed in a trusted environment, the VMM, while the monitored OS and users' applications, are run as a guest of the virtual machine. Recently, this research direction has been strongly encouraged

by the introduction of hardware extensions for the x86 architecture that simplify the development of virtual machine monitors [3, 4]. Since the hypervisor operates at a higher privilege level than the guest OS, it has complete control of the hardware, it can preemptively intercept events, it cannot be tampered by a compromised OS, and therefore it can be used to enforce stronger protection [5–9]. Advanced techniques, like the one used by Shadow Walker to hide malicious code, are defeated using out-of-the-box memory content scanners. Unfortunately, all the VMM-based solutions proposed in literature are based on the same assumption: they operate *proactively*. In other words, the hypervisor must be started before the guest OS and it must run until the guest terminates. Therefore, post-infection analysis of systems that were not running such VMM-based protections before an infection continues to be unsafe, because the malware and the tools used for the analysis run at the same privilege level.

In this paper we propose HyperSleuth, a tool that exploits the VMM extensions available nowadays (and typically unused) in commodity hardware, to securely perform *live forensic analyses* of potentially compromised production systems. HyperSleuth is executed on systems that are believed to be compromised, and obtains complete and tamper-resistant control over the OS, by running at "ring minus-one" (i.e., the hypervisor privilege level). HyperSleuth consists in (I) a tiny hypervisor that performs the analysis and (II) a secure loader that installs the hypervisor and verifies that its code is not tampered during installation. Like in virtualization-based malware, the *hypervisor is installed on-the-fly*: the alleged compromised host OS is transformed into a guest as it runs [10]. Since the hardware guarantees that the hypervisor is not accessible from the guest code, HyperSleuth remains persistent in the system for all the time necessary to perform the live analysis. On the contrary, other solutions proposed in literature for executing verified code in untrusted environments are not persistent and thus cannot guarantee that the verified code is not tampered when the execution of the untrusted code is resumed [11–13]. By providing a persistent trusted execution environment, HyperSleuth opens new opportunities for live and trusted forensic analyses, including the possibility to perform analyses that require to monitor the run-time behavior of the system. When the live analysis is concluded positively (e.g., no malicious program is found), HyperSleuth can be removed from the system and the OS, which was temporarily transformed into a guest OS, becomes again the host OS. As for the installation, the hypervisor is removed on-the-fly.

We developed a memory acquisition tool, a lie detector [6], and a system call tracer on top of HyperSleuth, to show how our hardware-supported VMM-based framework can be successfully used to gather volatile data even from production systems whose services cannot be interrupted. To experimentally demonstrate our claims about the effectiveness of HyperSleuth, we simulated two scenarios: a compromised production system running a heavy-loaded DNS server and a system infected by several kernel-level malware. We used HyperSleuth to dump the content of the physical memory of the former and to detect the malware in the latter. In the first case, HyperSleuth was able to dump the entire content of

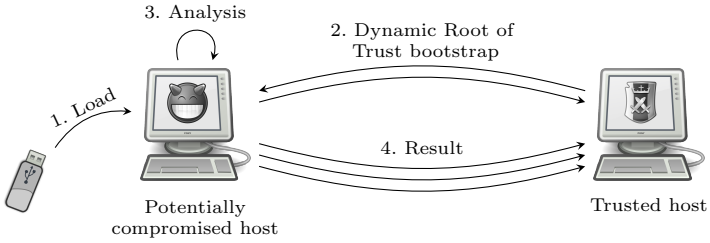

Fig. 1. Overview of HyperSleuth execution

the physical memory, without interrupting the services offered by the server. In the second case, HyperSleuth detected all the infections.

2 Overview

HyperSleuth should not be considered merely as a forensic tool, but rather as a framework for constructing forensic tools. Indeed, its goal is to provide a trusted execution environment for performing any *live forensic analysis* on production systems. More precisely, the execution environment in which a forensic analysis should be performed must guarantee four fundamental properties. First, the environment must guarantee a *tamper-proof* execution of the analysis code. That is, an attacker controlling the system cannot interfere with the analysis and cannot tamper the results. Second, it must be possible to perform an *a-posteriori bootstrap* of the trusted execution environment, even after the system has been compromised, and the bootstrap process itself must require no specific support from the system. Third, the trusted execution environment must be completely *transparent* to the system and to the attacker. Fourth, the trusted execution environment must be *persistent*. That is, the analysis performed in the trusted environment can be periodically interrupted, and the normal execution of the system resumed. Practically speaking, that allows to analyze an alleged compromised system without freezing it and without interrupting the services it provides. Moreover, such a property would allow to perform forensic analyses that require to monitor the run-time behavior of the system. As we will briefly see in the next sections, HyperSleuth fulfills all the aforementioned properties and can thus be used to safely analyze any compromised system that meets the requirements described in Section 2.3.

Figure 1 depicts the execution of HyperSleuth. HyperSleuth is installed and executed on demand (step 1 in Figure 1), only when there is a suspect that the host has been compromised, or in general when there is the necessity to perform a live forensic analysis. The execution is characterized by two phases. In the first phase (step 2 in Figure 1), HyperSleuth assumes complete control of the host and establishes a Dynamic Root of Trust (DRT). That is accomplished with the collaboration of a trusted host (located in the same local network). The trusted host is responsible for attesting that the DRT has been correctly established. In

the second phase (steps 3–4 in Figure 1), HyperSleuth performs a specific live forensic analysis and transmits the results of the analysis to the trusted host. Since the trusted host has a proof that the DRT has been correctly established and since, in turn, the DRT guarantees that the analysis code executes in the untrusted host untampered, the results of the analysis can be transitively considered authentic.

In the following, we briefly describe the architecture of HyperSleuth and how it manages to assume and maintain complete control of the untrusted host. Then, we describe the mechanism we use to bootstrap the dynamic root of trust, and, finally, we describe the assumptions and the threat model under which HyperSleuth runs.

2.1 HyperSleuth Architecture

HyperSleuth needs to be isolated from the host OS, to prevent any attack potentially originating from a compromised system. Simultaneously, HyperSleuth must be able to access certain resources of the host, to perform the requested forensic analysis, and to access the network to transmit the result to the trusted machine.

Figure 2 shows the position where HyperSleuth resides in the host. Since HyperSleuth needs to obtain and maintain complete control of the host and needs to operate with more privileges than the attacker, it resides at the lowest level: between the hardware and the host OS. HyperSleuth exploits hardware virtualization support available in commodity x86 CPUs [3,4] (which is typically unused). In other words, it executes at the privilege level of a Virtual Machine Monitor (VMM) and thus it has direct access to the hardware and its isolation from the host OS is facilitated by the CPU.

One of the peculiar features of HyperSleuth is the possibility to load and unload the VMM as the host runs. This hot-plug capability is indeed a very important feature: it allows to transparently take over an allegedly compromised system, turning, *on-the-fly*, its host OS into a guest one, and vice-versa at will. This is done without rebooting the system and thus preserving all those valuable run-time information that can allow to discover a malware infection or an intrusion. To do that, HyperSleuth leverages a characteristic of the hardware virtualization support available in x86 CPUs that allows to launch a VMM at any time, even when the host OS and users' applications are already running. Once the VMM is launched, the host becomes a *guest* of the VMM and the attacker loses her monopoly of the system and any possibility to tamper the execution of the VMM and the results of the forensic analysis.

The greyed portions in Figure 2 represent the trusted components in our system. During the launch, HyperSleuth assumes complete control of virtual memory management, to ensure that the host OS cannot access any of its private memory locations. Moreover, HyperSleuth does not trust any existing software component of the host. Rather, it contains all the necessary primitives to inspect directly the state of the guest and to dialog with the network card to transmit data to the trusted party.

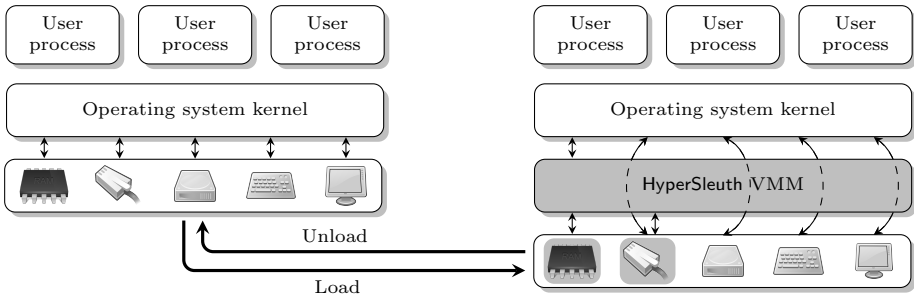

Fig. 2. Overview of HyperSleuth architecture

Depending on the type of forensic analysis, the analysis might be performed immediately after the launch, or it might be executed in multiple rounds, interleaved with the execution of the OS and users' applications. The advantage of the latter approach over the former is that the host can continue its normal activity while the analysis is being performed. Thus, the analysis does not result in a denial of service and can also target run-time evolving characteristics of the system. In both cases, when the analysis is completed, HyperSleuth can be disabled and even unloaded.

2.2 HyperSleuth Trusted Launch

HyperSleuth's launch process consists in enabling the VMM privilege level, in configuring the CPU to execute HyperSleuth code at this level, and in configuring the CPU such that all virtual memory management operations can be intercepted and supervised by the VMM. Unfortunately, an attacker could easily tamper the launch. For example, she could simulate a successful installation of the VMM and then transmit fake analysis results to the trusted host. This weakness stems from the fact that the launch process just described lacks an initial trusted component on which we can rely to establish the DRT.

The approach we use to establish the DRT is based on a primitive for tamper proof code execution. This primitive allows to create and to prove the establishment of a minimalistic trusted execution environment that guarantees that the code executed in this environment runs with maximum available privileges and that no attacker can manipulate the code before and during the execution. We use this primitive to create the environment to launch HyperSleuth and to prove to the trusted host that we have established the missing trusted component and that all subsequent operations are secured.

We currently rely on a pure software primitive that is based on a challenge and response protocol and involves an external trusted host [14]. Alternatively, a TPM-based hardware attestation primitive can be used for this purpose (e.g., Intel `senter` and AMD `skinit` primitives [3, 15]).

2.3 Requirements and Threat Model

Since HyperSleuth leverages hardware support for virtualization available in commodity CPUs, such support must be available on the system that must be analyzed[1]. To maximize the portability of HyperSleuth, we have designed it to only require first generation of hardware facilities for virtualization (i.e., HyperSleuth does not require extensions for MMU and I/O virtualization). Clearly, HyperSleuth cannot be used on systems on which virtualization support is already in use [16]. If a trusted VMM were already running on the host, the VMM could be used directly to perform the analysis. On the other side, if a malicious VMM were running on the host, HyperSleuth's trusted launch would fail.

In order to launch HyperSleuth some privileged instructions must be executed. That can be accomplished by installing a kernel driver in the target host. Note that, in the unlikely case of a damaged system that does not allow to load any kernel driver, alternative solutions for executing code in the kernel can be used (e.g., the page-file attack [10]).

The threat model under which HyperSleuth operates takes into consideration a very powerful attacker, e.g., an attacker with kernel-level privileges. Nonetheless, some assumptions were made while designing HyperSleuth. In particular, the attacker does not operate in system management mode, the attacker does not perform hardware-based attacks (e.g., a DMA-based attack), and the attacker does not leverage an external and more powerful host to simulate the bootstrap of the DRT. Some of these assumptions could indeed be relaxed by virtualizing completely I/O devices using either a pure-software approach or recent hardware support for devices virtualization (e.g., Intel VT-d), and by employing an hardware trusted platform for code attestation (e.g., TPM), keeping HyperSleuth a secure and powerful framework for performing forensic analysis of live data.

3 Implementation

The core of HyperSleuth is a minimalistic virtual machine monitor that is installed on the host while the OS and users' applications are already running. We achieve this goal by exploiting hardware support for virtualization available in modern x86 CPUs. In this Section we describe how we have implemented HyperSleuth on a system with an Intel x86 CPU with VT-x extensions.

3.1 Intel VT-x

Before presenting the details of HyperSleuth VMM implementation, we give a brief overview of the hardware virtualization technology available in Intel x86 CPUs, called VT-x. AMD technology, named SVM, is very similar and differs mostly in terms of terminology.

[1] Although nowadays all consumer CPUs come with hardware support for virtualization, in order to be usable, the support must be enabled via the BIOS. At the moment we do not know how many manufactures enable the support by default.

Intel VT-x separates the CPU execution into two modes of operation: *VMX root mode* and *VMX non-root mode*. The VMM and the guest (OS and applications) execute respectively in root and non-root modes. Software executing in both modes can operate in any of the four privilege levels that are supported by the CPU. Thus, the guest OS can execute at the highest CPU privilege and the VMM can supervise the execution of the guest without any modification of the guest. When a VMM is installed, the CPU switches back and forth between non-root and root mode: the execution of the guest might be interrupted by an *exit* to root mode and subsequently resumed by an *enter* to non-root mode. After the launch, the VMM execution is never scheduled and exits to root-mode are the only mechanism for the VMM to regain the control of the execution. Like hardware exceptions, exits are events that block the execution of the guest, switch from non-root mode to root mode, and transfer the control to the VMM. However, differently from exceptions, the set of events triggering exits to root mode can be configured dynamically by the VMM. Examples of exiting events are exceptions, interrupts, I/O operations, and the execution of privileged instructions that access control registers or descriptor tables. Exits can also be requested explicitly by the guest through a *VMM call*. Exits are handled by a specific VMM routine that eventually executes an enter to resume the execution of the guest. The state of the CPU at the time of an exit and of an enter is stored in a data structure called Virtual Machine Control Structure, or *VMCS*. This structure also controls the set of events triggering exists and the state of the CPU for executing in root-mode.

In the typical deployment, the launch of the VMM consists of three steps. First, the VMX root-mode is enabled. Second, the CPU is configured to execute the VMM in root-mode. Third, the guests are booted in non-root mode. However, Intel VT-x allows to launch a VMM at any time, thus giving the ability to transform a running host into a guest of a VMM. The procedure for such a delayed launch is the same as the one just described, with the exception of the third step. The state of the CPU for non-root mode is set to the exact same state of the CPU preceding the launch, such that, when the launch is completed, the execution of the OS and its applications resumes in non-root mode. The inverse procedure can be used to unload the VMM, disable VMX root-mode, and give back full control of the system to the OS.

3.2 HyperSleuth VMM

HyperSleuth can be loaded at any time by exploiting the delayed launch feature offered by the CPU. Figure 3 shows a simplified memory layout after the launch of HyperSleuth. The environment for non-root mode, in which the OS and users' application are executed, is left intact. The environment for root mode instead is created during the launch and maintained isolated by the VMM. The VMCS controls the execution contexts of both root and non-root modes. In the following paragraphs we describe in details the steps required to launch the VMM, to recreate the environment for running the OS and users' applications, and to enforce the isolation of root-mode from non-root mode.

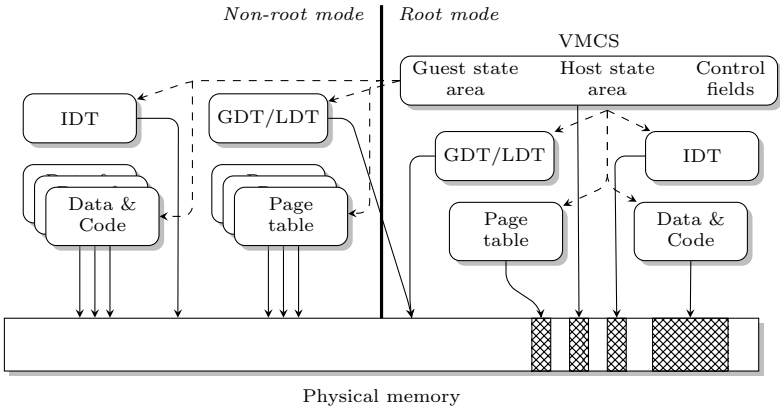

Fig. 3. Memory layout after the launch of HyperSleuth; ⇢ denotes the CPU contexts stored in the VMCS, ⟶ denotes physical memory mappings, and ⊠ denotes the physical memory locations of the VMM that must not be made accessible to the guest.

VMM Launch. To launch HyperSleuth VMM in a running host we perform the following operations. First, we allocate a fixed-size chunk of memory to hold the data and code of the VMM. Second, we enable VMX root-mode. Third, we create and initialize the VMCS. Fourth, we resume the normal execution of the guest by entering non-root mode.

When, at the end of the launch, the CPU enters non-root mode, it loads the context for executing the guest from the *guest-state area* of the VMCS. The trick to load the VMM without interrupting the execution of the OS and users' applications is to set, in the VMCS, the context for non-root mode to the same context in which the launch was initiated. The context in which the VMM executes is instead defined by the *host-state area* of the VMCS. Like during an enter, the CPU loads the context from the VMCS during an exit. The context is created from scratch during the launch and the host-state area is configured accordingly. In particular, we create and register a dummy Interrupt Descriptor Table (to ignore interrupts that might occur during switches between the two VMX modes), we register the Global and Local Descriptor Tables (we use the same tables used in non-root mode), we register the address of the VMM entry point (i.e., the address of the routine for handling exits), and we assign the stack.

The set of events that trigger exits to root-mode are defined in the *execution control fields* of the VMCS. The configuration of these fields depends on the type of the forensic analysis we want to perform and can be changed dynamically.

VMM Trusted Launch. Although on the paper the launch of the VMM appears a very simple process, it requires to perform several operations. Such operations must be performed atomically, otherwise a skilled attacker may interfere with the whole bootstrap process and tamper VMM code and data. To maximize HyperSleuth portability, we decided to address this problem using a software-based primitive for tamper-proof code execution. The primitive we rely

on is thoroughly described in [14]. In a few words, the primitive is based on a challenge-response protocol and a checksum function. The trusted host issues a challenge for the untrusted system and the challenge consists in computing a checksum. The result of the checksum is sent back to the trusted host. A valid checksum received within a predefined time is the proof that a Trusted Computing Base (TCB) has been established on the untrusted system. The checksum function is constructed such that the correct checksum value can be computed in time only if the checksum function and the code for launching the VMM are not tampered, and if the environment in which the checksum is computed and in which the VMM launch will be performed guarantees that no attacker can interrupt the execution and regain the control of the execution before the launch is completed. Practically speaking, the correct checksum will be computed in time only if the computation and the launch are performed with kernel privileges, with interrupts disabled, and no VMM is running.

MMU Virtualization. In order to guarantee complete isolation of the VMM from the guest, it is essential to ensure that the guest cannot access any of the memory pages in use by the VMM (i.e., the crosshatched regions in Figure 3). However, to perform any useful analysis, we need the opposite to be possible.

Although modern x86 CPUs provide hardware support for MMU virtualization, we have opted for a software-based approach to maximize the portability of HyperSleuth. The approach we use is based on the assumption that the direct access to physical memory locations is not allowed by the CPU (with paging enabled) and that physical memory locations are referenced through virtual addresses. The CPU maintains a mapping between virtual and physical memory locations and manages the permissions of these locations through page tables. By assuming the complete control of the page tables, the VMM can decide which physical locations the guest can access. To do that, the VMM maintains a *shadow page table* for each page table used by the guest, and tricks the guest into using the shadow page table instead of the real one [17].

A shadow page table is a clone of the original page table and is used to maintain a different mapping between virtual and host physical addresses and to enforce stricter memory protections. In our particular scenario, where the VMM manages a single guest and the OS has already filled the page tables (because the VMM launch is delayed), the specific duty of the shadow page table is to maintain as much as possible the original mapping between virtual and physical addresses and to ensure that none of the pages assigned to the VMM is mapped into a virtual page accessible to the guest. As described in Section 4, we also rely on the shadow page table to restrict and trap certain memory accesses to perform the live forensic analysis. The algorithm we currently use to maintain the shadow page tables trades off performance for simplicity and is based on tracing and simulating all accesses to tables.

Unrestricted Guest Access to I/O Devices. In the typical deployment, physical I/O devices connected to the host are shared between the VMM and one or more guests. In our particular scenario, instead, there is no need to share any I/O device between the guest and the VMM: HyperSleuth executes batch and

interacts only with the trusted host via network. Thus, the guest can be given direct and unrestricted access to I/O devices. Since the OS runs in non-root mode, unmodified, and at the highest privilege level, it is authorized to perform I/O operations, unless the VMM configures the execution control fields of the VMCS such that I/O operations cause exits to root-mode. By not doing so, the VMM allows the guest OS to perform unrestricted and direct I/O. This approach simplifies drastically the architecture of the VMM and, most importantly, allows the OS to continue to perform I/O activities exactly as before, without any additional overhead.

Direct Network Access. HyperSleuth relies on a trusted host to bootstrap the dynamic root of trust and to store the result of the analysis. Since we are assuming that no existing software component of the host can be trusted, the only viable approach to communicate securely over the network is to dialog directly with the network card. For this reason, HyperSleuth contains a minimalistic network driver that supports the card available on the host. All the data transmitted over the network is encapsulated in UDP packets. Packets are signed and encrypted automatically by the driver using a pre-shared key, which we hardcode in HyperSleuth just before the launch.

As described in the previous paragraph, HyperSleuth does not virtualize hardware peripherals, but it lets the guest to access them directly. Thus, the network card must be shared transparently with the guest. In other words, to avoid interferences with the network activity of the guest, HyperSleuth must save and restore the original state of the card (i.e., the content of PCI registers), respectively before and after using the network. To transmit a packet the driver writes the physical address and the size of the packet to the appropriate control registers of the device. The driver then polls the status register of the device until the transmission is completed. Polling is used because, for simplicity, we execute all VMM code with interrupts disabled. Packets reception is implemented in the same way.

VMM Removal. HyperSleuth can be completely removed from the system at the end of the analysis. The removal essentially is the opposite process of the launch. First, we disable VMX root-mode. Second, we deallocate the memory regions assigned to the VMM (e.g., the Interrupt Descriptor Table, the stack, and the code). Third, we update the context of the CPU such that the OS and users' applications can resume their normal execution. More precisely, we set the context to that stored in the guest-state area of the VMCS, which reflects the context of the CPU in non-root mode when the last exit occurred. Fourth, we transfer the execution to a small snippet of code that deallocates the VMCS and then transfers the control to where the execution was interrupted in non-root mode.

4 Live Forensic Analysis

HyperSleuth operates completely in batch mode. The only user action required is to copy an executable on the system to be analyzed and to fire its execution.

This executable is a loader that establishes the dynamic root of trust by creating a tamper-proof execution environment and by using this environment to launch the VMM. Note that, the loader is removed from the memory and the disk to prevent malicious software to detect its presence. Once launched, the VMM performs the forensic analysis, transmits the results to the trusted hosts and then removes itself.

Although HyperSleuth VMM is completely transparent to the OS and users' applications and it is removed after the end of the analysis, the launch of the VMM is a slightly invasive process. Indeed, it requires to execute the loader that in turn loads a kernel driver (to launch the VMM) and might start other additional in-guest utilities. Our claim is that, considered the valuable volatile information HyperSleuth can gather from the system, the little modifications its installation produces to the state of the system are an acceptable compromise. After all, no zero invasive solution for *a posteriori* forensic analysis exists.

Currently, HyperSleuth supports three live forensic applications: a lazy physical memory dumper, a lie detector, and a system call tracer. Clearly, all these analyses could be performed also without the need of a dynamic root of trust and the VMM. Indeed, there are several commercial and open source applications with the same capabilities available, but, by operating at the same privilege level of the OS kernel to analyze, they can easily be tampered by an attacker (with the same privileges), and cannot thus provide the safety guarantees offered by HyperSleuth.

4.1 Physical Memory Dumper

Traditional approaches for dumping the content of the physical memory are typically based on kernel drivers or on FireWire devices. Unfortunately, both approaches have a major drawback that limits their applicability to non production systems. Dumping the content of the physical memory is an operation that should be performed atomically, to guarantee the integrity of the dumped data. Failing to achieve this would, in fact, enable an attacker to make arbitrary modification to the content of the memory, potentially hampering any forensic analysis of live data. On the other side, if the dump is performed atomically, the system, and the services the system provides, will be blocked for the entire duration of the dump. That is not desirable, especially if there is only a marginal evidence that the system has been compromised. Being the dump very time consuming, the downtime might be economically very expensive and even dangerous.

To address this problem, we exploit HyperSleuth's persistent trusted execution environment to implement a new approach for dumping lazily the content of the physical memory. This approach guarantees that the state of the physical memory dumped corresponds to the state of the memory *at the time the dump is requested*. That is, no malicious process can "clean" the memory after HyperSleuth has been installed. Moreover, being performed lazily, the dump of the state of the memory does not monopolize the CPU and does not interrupt the execution of the processes running in the system. In other words, HyperSleuth

```
1   switch (VMM exit reason)
2     case CR3 write:
3        Sync PT and SPT
4        for (v = 0; v < sizeof(SPT); v++)
5           if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
6              SPT[v].Writable = 0;
7
8     case Page fault: // 'v' is the faulty address
9        if (PT/SPT access)
10          Sync PT and SPT and protect SPTEs if necessary
11       else if (write access && PT[v].Writable)
12          if (!DUMPED[PT[v].PhysicalAddress])
13             DUMP(PT[v].PhysicalAddress);
14          SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
15       else
16          Pass the exception to the OS
17
18    case Hlt:
19       for (p = 0; p < sizeof(DUMPED); p++)
20          if (!DUMPED[p])
21             DUMP(p); DUMPED[p] = 1;
22             break;
```

Fig. 4. Algorithm for lazy dump of the physical memory

allows to dump the content of the physical memory even of a production system without causing any outage of the services offered by the system.

The dump of the memory is transmitted via network to the trusted host. Each page is fragmented, to fit the MTU of the channel, and labelled. The receiver reassembles the fragments and reorders the pages to reconstruct the original bitstream image of the physical memory. To ease further analysis, the image produced by HyperSleuth is compatible with off-the-shelf tools for memory forensic analysis (e.g., Volatility [18]).

The algorithm we developed for dumping lazily the content of the physical memory is partially inspired by the technique used by operating systems for handling shared memory and known as *copy-on-write*. The rationale of the algorithm is that the dump of a physical memory page can be safely postponed until the page is accessed for writing. More precisely, the algorithm adopts a combination of two strategies to dump the memory: *dump-on-write* (DOW), and *dump-on-idle* (DOI). The former permits to dump a page before it is modified by the guest; the latter permits to dump a page when the guest is idle. Note that the algorithm assumes that the guest cannot access directly the physical memory. However, an attacker could still program a hardware device to alter the content of the memory by performing a DMA operation. In our current threat model we do not consider DMA-based attacks.

Figure 4 shows the pseudo-code of our memory dumper. Essentially the VMM intercepts three types of events: updates of the page table address, page-fault exceptions, and CPU idle loops. The algorithm maintains a map of the physical pages that have already been dumped (`DUMPED`) and leverages the shadow page table (`SPT`) to enforce stricter permissions than the ones specified in the real page table (`PT`) currently used by the system. When the page table address (stored in the `CR3` register) is updated, typically during a context switch, the algorithm synchronizes the shadow page table and the page table (line 3). Subsequently,

all the entries of the shadow page table mapping physical not yet dumped pages are granted read-only permissions (lines 4–6). Such a protection ensures that all the memory accesses performed by the guest OS for writing to any virtual page mapped into a physical page that has not been dumped yet results in a page fault exception. The VMM intercepts all the page fault exceptions for keeping the shadow page table and the real page table in sync, for reinforcing our write protection after every update of the page table (lines 9–10), and also for intercepting all write accesses to pages not yet dumped (lines 11–14). The latter type of faults are characterized by a write access to a non-writable virtual page that is marked as writable in the real page table. If the accessed physical page has not been dumped yet, the algorithm dumps the page and flags it as such. All other types of page fault exceptions are delivered to the guest OS that will manage them accordingly. Finally, the VMM detects CPU idle loops by intercepting all occurrences of the `hlt` instruction. This instruction is executed by the OS when there is no immediate work to be done, and it halts the CPU until an interrupt is delivered. We exploit these short idle periods to dump the pending pages (lines 19–22). It is worth noting that a loaded system might enter very few idle loops. For this reason, at every context switch we check whether the CPU has recently entered the idle loop and, if not, we force a dump of a small subset of the pending pages (not shown in the figure).

4.2 Lie Detector

Kernel-level malware are particularly insidious as they operate at a very high privilege level and can, in principle, hide any resource an attacker wants to protect from being discovered (e.g., processes, network communications, files). Different techniques exist to achieve such a goal (see [1]), but all of them aim at forcing the OS to lie about its state, eventually. Therefore, the only effective way to discover such liars is to compare the state of the system perceived from the system itself with the state of the system perceived by a VMM. Unfortunately, so far lie detection has been possible only using a traditional VMM and thus it has not been applicable on production systems not already deployed in virtual machine environments. On the other hand, HyperSleuth's hot-plug capability of securely migrating a host OS into a guest one (and vice-versa) on-the-fly makes it a perfect candidate for detecting liars in production systems that had not been deployed in virtual machine environments since the beginning.

To this end, besides launching the VMM, HyperSleuth loader runs a simple in-guest utility that collects detailed information about the state of the system and transmits its output to the trusted host. This utility performs the operations typically performed by system tools to display information about the state of the system and intentionally relies on the untrusted code of the OS. The intent is to trigger the malicious code installed by the attacker to hide any malicious software component or activity. For example, this utility collects the list of running processes, active networks connections, loaded drivers, open files and registry keys, and so on. At the end of its execution, the utility performs a VMM call to transfer the execution to the HyperSleuth VMM. At this point the VMM

collects the same information through OS-aware inspection. That is, the VMM does not rely on any untrusted code of the system, but rather implements its own primitives for inspecting the state of the guest and, when possible, offers multiple primitives to inspect the state of the same resource. For example it offers primitives to retrieve the list of running processes/threads, each of which relies on a different data structure available in the kernel. Finally, the trusted host compares the views provided by the in-guest utility and the VMM.

Since the state of the system changes dynamically and since the in-guest utility and the VMM does not run simultaneously, we repeat the procedure multiple times, with a variable delay between each run to limit any measurement error.

4.3 System Call Tracer

System calls tracing has been widely recognized as a way to infer, observe, and understand the behavior of processes [19]. Traditionally, system calls were invoked by executing software interrupt instructions causing a transition from user-space to kernel-space. Such user-/kernel-space interactions can be intercepted by HyperSleuth, as interrupt instructions executed by the guest OS in VMX non-root mode cause an exit to VMX root mode, i.e., to the VMM.

Alternative and more efficient mechanisms for user-/kernel-space interactions have been introduced by CPU developers, recently. Unfortunately, Intel VT-x does not support natively the tracing of system calls invoked through the `sysenter/sysexit` fast invocation interface used by modern operating systems. The approach we use to trace system calls is thus inspired by Ether [5]. System calls are intercepted through another type of exits: synthetic page fault exceptions. All system calls invocations go through a common gate, whose address is defined in the `SYSENTER_EIP` register. We shadow the value of this register and set the value of the shadow copy to the address of a non-existent memory location, such that all system calls invocations result in a page fault exception and in an exit to root mode. The VMM can easily detect the reason of the fault by inspecting the faulty address. When a system call invocation is trapped by the VMM, it logs the system call and then resumes the execution of the guest from the real address of `SYSENTER_EIP`. To intercept returns from system calls we mark the page containing the return address as not accessible in the shadow page table. The log is transmitted via network to the trusted host.

5 Experimental Evaluation

We implemented a prototype of the VMM and of the routines for the three analyses described in Section 4. Our current implementation of HyperSleuth is specific for the Microsoft Windows XP (32-bit) operating system. While the core of HyperSleuth is mostly OS-independent, the routines for the analysis (e.g., the enumeration of running processes and of active network connections) are OS-dependent and may require to be slightly adapted to provide support for different operating systems.

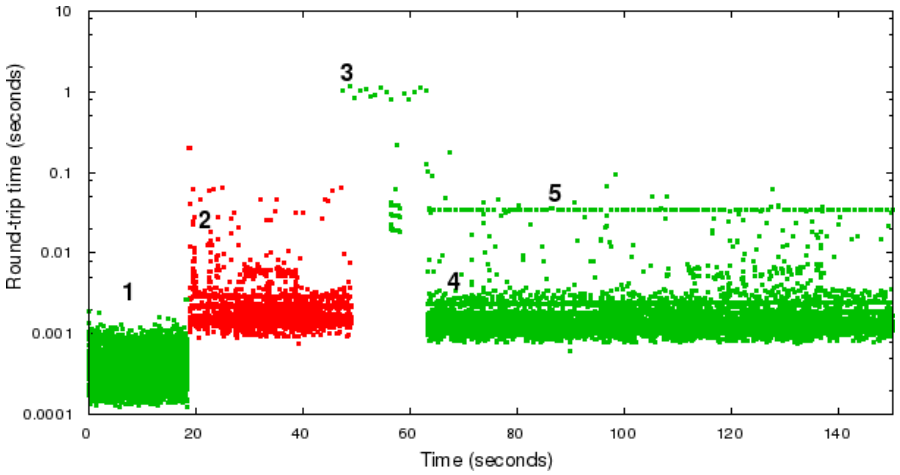

Fig. 5. Round-trip time of the queries performed against the compromised production DNS server before (1) and after (2) the launch of HyperSleuth and (3–5) during the lazy dump of the physical memory (the scale of the ordinate is logarithmic).

In this section we discuss the experimental results concerning the launch of HyperSleuth, the lazy physical memory dumper, and the lie detector. To this end, we simulated the compromised production system using an Intel Core i7, with 3GB RAM, and a Realtek RTL8139 100Mbps network card. Note that we disabled all cores of the CPU but one, since the VMM currently supports a single core. We simulated the trusted host using a laptop. We used the trusted host to attest the correct establishment of the dynamic root of trust and to collect and subsequently analyze the results of the analysis.

5.1 HyperSleuth Launch and Lazy Dump of the Physical Memory

To evaluate the cost of launching HyperSleuth, the base overhead of the VMM, and the cost of the lazy physical memory dumper we simulated the following scenario. A production DNS server was compromised and we used HyperSleuth to dump the entire content of the physical memory when the server was under the heaviest possible load. We used an additional laptop, located on the same network, to flood the DNS server with queries and to measure the instantaneous round-trip time of the queries. About 20 seconds after we started the flood, we launched HyperSleuth; 25 seconds later we started to dump the content of the memory.

Figure 5 summarizes the results of our experiments. The graph shows the round-trip time of the queries sent to the compromised DNS server over time. For the duration of the experiment, the compromised machine was able to handle all the incoming DNS queries, and no query timed out. Before launching HyperSleuth the average round-trip time was $\sim 0.34ms$ (mark 1 in Figure 5). Just after the launch, we observed an initial increase of the round-trip time to

about $0.19s$ (mark 2 in Figure 5). This increase was caused by the bootstrap of the dynamic root of trust and then by the launch of the VMM, which must be performed atomically. After the launch, the round-trip time quickly stabilized around $1.6ms$, less than five times the round-trip time without the VMM. The overhead introduced by the VMM was mostly caused by the handling of the shadow page table. When we started the dump of the physical memory we observed another and steeper peak (mark 3 in Figure 5). We were expecting this behavior since there are a lot of writable memory pages that are frequently accessed (e.g., the stack of the kernel and of the user-space processes and the global variables of the kernel) and that, most likely, are written each time the corresponding process is scheduled. Thus, the peak was caused by the massive number of write accesses to pages not yet dumped. A dozen of seconds later the round-trip time stabilized again around $1.6ms$ (mark 4 in Figure 5). That corresponds to the round-trip time observed before we started the dump. Indeed, the most frequently written pages were written immediately after the dump was started, and the cost of the dump of a single page was much less than the round-trip time and was thus unnoticeable. The regular peaks around $32ms$ about every second (mark 5 in Figure 5) were instead caused by the periodic dump of non-written pages. Since the system was under heavy load, it never entered an idle loop. Thus, the dump was forced after every second of uninterrupted CPU activity. More precisely, the dumper was configured to dump 64 physical pages about every second. Clearly, the number of non-written pages to be dumped when either the system enters the idle loop, or the duration of uninterrupted CPU activity hits a certain threshold, is a parameter that can be tuned accordingly to the urgency of the analysis, to how critical the system is, and to the throughput of the network.

In conclusion, the dump of the whole physical memory of the system (3GB of RAM), in the setting just described, required about 180 minutes and the resulting dump could be analyzed using an off-the-shelf tool, such as Volatility [18]. The total time could be further decreased by increasing the number of physical pages dumped periodically, at the cost of a higher average round-trip time. It should also be pointed out that, on a 1Gbps network, we could increase the number of physical pages dumped every second to 640, without incurring in any additional performance penalty. In this case, the whole physical memory (3GB) would be dumped in just $\sim$ 18 minutes. It is important to remark that although HyperSleuth, and in particular the algorithm for dumping lazily the memory, introduces a non-negligible overhead, we were able to dump the entire content of the memory without interrupting the service (i.e., no DNS query timed out). On the other hand, if the memory were dumped with traditional (atomic) approaches the dump would require, in the ideal case, about 24 seconds, 50 seconds, and 4 minutes respectively on a 1Gbps network, on a 480Mbps FireWire channel, and on a 100Mbps network (these estimations are computed by dividing the maximum throughout of the media by the amount of data to transmit). In these cases, the production system would have not been able to handle any incoming request, for the entire duration of the dump.

Sample	Characteristics	Detected?
FU	DKOM	✓
FUTo	DKOM	✓
HaxDoor	DKOM, SSDT hooking, API hooking	✓
HE4Hook	SSDT hooking	✓
NtIllusion	DLL injection	✓
NucleRoot	API hooking	✓
Sinowal	MBR infection, Run-time patching	✓

Table 1. Results of the evaluation of HyperSleuth's lie detector with seven different malware (all equipped with a root-kit component)

5.2 Lie Detection

Table 1 summarizes the results of the experiments we performed to assess the efficacy of the lie detection module. To this end, we used seven malware samples, each of which included a root-kit component to hide the malicious activity performed on the infected system. We used HyperSleuth's lie detector to detect the hidden activities. The results testify that our approach can be used to detect both user- and kernel-level root-kits.

For each malware sample we proceeded as follows. First, we let the malware infect the untrusted system. Then, we launched HyperSleuth on the compromised host and triggered the execution of the lie detector. The module performed the analysis, first by leveraging the in-guest utility, and then by collecting the same information directly from the VMM through OS-aware inspection. The results were sent separately to the trusted host. On the trusted host we compared the two views of the state of the system and, in all cases, we detected some discrepancies between the two. These discrepancies were all caused by lies. That is, the state visible to the in-guest utility was altered by the root-kit, while the state visible to HyperSleuth VMM was not.

As an example, consider the FUTo root-kit. This sample leverages direct kernel object manipulation (DKOM) techniques to hide certain kernel objects created by the malware (e.g., processes) [1]. Our current implementation of the lie detector counteracts DKOM through a series of analyses similar to those implemented in RAIDE [20]. Briefly, those analyses consist in scanning some internal structures of the Windows kernel that the malware must leave intact in order to preserve its functionalities. Thus, when we compared the trusted with the untrusted view of the state of the system we noticed a process that was not present in the untrusted view produced by the in-guest utility. Another interesting example is NucleRoot, a root-kit that hooks Windows' System Service Descriptor Table (SSDT) to intercept the execution of several system calls and to filter out their results, in order to hide certain files, processes, and registry keys. In this case, by comparing the two views of the state of the system, we observed that some registry keys related to the malware were missing in the untrusted view. Although we have not yet any empirical proof, we speculate the even rootkits like Shadow Walker [2] would be detected by our lie detector

since our approach allows to inspect the memory directly, bypassing a malicious page-fault handler and bogus TLBs' entries.

6 Discussion

We presented HyperSleuth from a technical prospective. The decisions we made in designing and implementing HyperSleuth were mostly motivated by the intent of minimizing the dependencies on the hardware and of maximizing the portability. Therefore, we always opted for pure software-based approaches (e.g., to secure the launch of the VMM and to virtualize the MMU), whenever possible. However, since HyperSleuth is a framework for performing live forensic analyses, it is important to reason about its probatory value. From such a prospective, we must take into account that the trustworthiness of the results of the analyses depends on the trust people have in the tool that generated the results. To strengthen its probatory value, all HyperSleuth's components should be verified in order to prove that their code meets all the expectations [21]. At this aim, in the future we plan to further decrease the size of HyperSleuth's code base in order to ease its verifiability (e.g., by leveraging hardware-based attestation solutions, such as the TPM).

HyperSleuth's effectiveness depends on the impossibility to detect its presence from the guest. Although the VMM is completely isolated from the guest, the malware might attempt to detect HyperSleuth by trying to install another VMM. One approach to contrast such attempts is to let the malware believe that virtualization support is not available at all.

7 Related work

The idea of leveraging a virtual machine monitor to perform sophisticated runtime analyses, with the guarantee that the results cannot be tampered by a malicious attacker, has already been widely explored in the literature. Garfinkel *et al.* were the first to propose to use a VMM to perform OS-aware introspection [6], and subsequently the idea was further elaborated [5, 22]. Other researchers instead proposed to use a VMM to protect the guest OS from attacks by supervising its execution, both with a software-based VMM [8] and by leveraging hardware support for virtualization [9]. Similar ideas were also suggested by other authors [7, 23]. In [24] Chen *et al.* proposed a solution to protect applications' data even in the presence of a compromised operating system. More recently, Vasudevan *et al.* proposed XTREC, a lightweight framework to record securely the execution control flow of all code running in an untrusted system [25]. Unfortunately, in order to guarantee that the analyses they perform cannot be tampered by an attacker, all the aforementioned solutions must take control of the system before the guest is booted, and cannot be removed until the guest is shut down. On the contrary, HyperSleuth can be installed as the compromised system runs, and, when the analyses are completed, it can be removed on-the-fly. The idea to take advantage of the possibility to install a VMM on a running system was also

sketched in [26], and later investigated in our previous research work to realize HyperDbg, a transparent kernel-level debugger [27].

Several researchers proposed to use VMMs to implement malware that are particularly hard to detect and to eradicate. SubVirt was one of the first prototypes that employed this technique [28]. However, being implemented using a software-based VMM, the installation of Subvirt required to reboot the machine, and the malware also introduced a noticeable run-time overhead in the infected target. Later, the Blue Pill malware started to exploit the hardware-assisted supports for virtualization to implement an efficient VMM-based malware that is able to infect a machine as it runs, without the need for reboot [10]. HyperSleuth was inspired by this malware.

8 Conclusion

We presented HyperSleuth, a framework for constructing forensic tools that leverages the virtualization extensions provided by commodity hardware to guarantee that the results of the analyses cannot be altered, even by an attacker with kernel-level privileges. HyperSleuth consists in a tiny hypervisor that is installed on a potentially compromised system as it runs, and a secure loader that installs the hypervisor and verifies its integrity. We developed a proof-of-concept prototype of HyperSleuth and, on top of it, we implemented three forensic analysis applications: a lazy physical memory dumper, a lie detector, and a system call tracer. Our experimental evaluation testified the effectiveness of the proposed approach.

References

1. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional (2005)
2. Sparks, S., Butler, J.: Shadow Walker. Raising The Bar For Windows Rootkit Detection. Phrack Magazine (Vol. 11, No. 63) (2005)
3. AMD, Inc.: AMD Virtualization `www.amd.com/virtualization`.
4. Intel Corporation: Intel Virtualization Technology `http://www.intel.com/technology/virtualization/`.
5. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. (2008)
6. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the Network and Distributed Systems Security Symposium, The Internet Society (2003)
7. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An Architecture for Secure Active Monitoring Using Virtualization. In: Proceedings of the IEEE Symposium on Security and Privacy. (2008)
8. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection. (2008)

9. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: Proccedings of the ACM Symposium on Operating Systems Principles, ACM (2007)
10. Rutkowska, J.: Subverting Vista Kernel For Fun And Profit. Black Hat USA (2006)
11. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for tcb minimization. In: Proceedings of the ACM European Conference in Computer Systems. (2008)
12. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In: Proceedings of ACM Symposium on Operating Systems Principles. (2005)
13. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: Swatt: Software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy. (2004)
14. Martignoni, L., Paleari, R., Bruschi, D.: Conqueror: tamper-proof code execution on legacy systems. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment. Lecture Notes in Computer Science (2010)
15. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press (2009)
16. Carbone, M., Zamboni, D., Lee, W.: Taming virtualization. IEEE Security and Privacy **6**(1) (2008)
17. Smith, J.E., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann (2005)
18. Volatile Systems LLC: Volatility (`http://www.volatilesystems.com/`).
19. Forrest, S., Hofmeyr, S.R., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the IEEE Symposium on Security and Privacy. (1996)
20. Butler, J., Silberman, P.: RAIDE: Rookit analysis identification elimination. In: Black Hat USA. (2006)
21. Franklin, J., Seshadri, A., Qu, N., Datta, A., Chaki, S.: Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical report, Carnegie Mellon University (2008)
22. Jiang, X., Wang, X.: "out-of-the-box" monitoring of VM-based high-interaction honeypots. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection. (2007)
23. Sharif, M., Lee, W., Cui, W., Lanzi, A.: Secure In-VM Monitoring Using Hardware Virtualization. In: Proceedings of the ACM Conference on Computer and Communications Security. (2009)
24. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. Operating Systems Review **42**(2) (2008)
25. Perrig, A., Gligor, V., Vasudevan, A.: XTREC: secure real-time execution trace recording and analysis on commodity platforms. Technical report, Carnegie Mellon University (2010)
26. Sahita, R., Warrier, U., Dewan, P.: Dynamic software application protection. Technical report, Intel Corporation (2009)
27. Fattori, A., Paleari, R., Martignoni, L., Monga, M.: HyperDbg: a fully transparent kernel-level debugger `http://code.google.com/p/hyperdbg/`.
28. King, S.T., Chen, P.M., Wang, Y.M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: Proceedings of IEEE Symposium on Security and Privacy. (2006)